Designing Parallel Data Processing for Large-Scale Sensor Orchestration

Milan Kabáč
Inria Bordeaux
Bordeaux, France
email: milan.kabac@inria.fr

Charles Consel
Bordeaux Institute of Technology
Bordeaux, France
email: charles.consel@inria.fr

***Abstract*—Masses of sensors are being deployed at the scale of cities to manage parking spaces, transportation infrastructures to monitor traffic, and campuses of buildings to reduce energy consumption. These large-scale infrastructures become a reality for citizens via applications that orchestrate sensors to deliver high-value, innovative services. These applications critically rely on the processing of large amounts of data to analyze situations, inform users, and control devices.**

This paper proposes a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of data. Specifically, an application design exposes declarations that are used to generate a programming framework based on the MapReduce programming model.

We have developed a prototype[1] of our approach, using Apache Hadoop. We applied it to a case study and obtained significant speedups by parallelizing computations over twelve nodes. In doing so, we demonstrate that our design-driven approach allows to abstract over implementation details, while exposing architectural properties used to generate high-performance code for processing large datasets.

I. Introduction

Modern ubiquitous computing systems take the form of wide-area infrastructures, populating a variety of environments with functionality-rich sensors. These smart environments include wide-area transportation management [1], [2] and large-scale smart parking systems [3], [4]. The emergence of smart environments validates large-scale sensor infrastructures as robust platforms for delivering innovative services to citizens.

Nevertheless, the successful adoption of these infrastructures critically relies on the ability to develop services. Currently, software development in this domain lacks programming models and methodologies to address key domain-specific challenges. In particular, masses of sensors produce large amounts of data that require to be analyzed efficiently to render high-value services to citizens and operators of smart environments. When considering tens of thousands of measurements, possibly accumulated over a period of time, processing becomes a critical issue. In fact, the amount of data to be processed and the requirements of the applications to be developed may necessitate *parallel processing* [5]. For example, as cars rush into a city in the morning, drivers should receive up-to-date information about space availability in parking lots, even if this involves processing massive amounts of data repeatedly. When efficiency is paramount, it is a key challenge to develop an orchestrating application that exploits properties about the sensors, optimizes the strategies to collect sensor measurements, and crunches large amounts of data.

Existing approaches dedicated to big data processing provide limited ways to combine data processing strategies with the application logic. Apache Pig [6] and Hive [7] require developers to describe data processing in SQL-like query languages with limited support for user-defined functions. Language libraries, such as FlumeJava [8] allow developers to implement data processing via high-level language abstractions. These approaches provide data flow expressions and a set of rich data types to implement data processing. Developers still need to decide when and where data processing occurs, as well as how intermediate computations are combined. In the case of large-scale orchestration, applications may have to analyze sensor data a number of times using different algorithms, or combine them. These needs put an additional burden on developers since they have to introduce boilerplate code to separate library-specific code from the main application logic, interconnect and coordinate computations, store intermediate results, *etc.*

This paper proposes a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of data. In doing so, we extend our previous work on a design language dedicated to orchestrating sensors, named DiaSwarm [9], which did not address high-performance data processing. Our new approach provides the developer with declarations expressing when and where data processing occurs. The application design then compiles into a programming framework, based on the MapReduce programming model. This framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

[1] `http://phoenix.inria.fr/software/diaswarm`

A. Our contributions

High-level parallel processing model.
Our approach allows the developer to program against a framework based on the *MapReduce* programming model [10], [11]. In doing so, the developer uses a well-proven approach to processing large datasets, based on a parallel implementation. We illustrate our approach with a case study of a parking management system.

A generative programming approach.
The generated parallel-processing programming frameworks have a carefully structured data and control flow, which enables data processing to be implemented efficiently. Our compiler generates programming frameworks that rely on the MapReduce model, exposing structural parallelism of the implementation. This strategy allows to cope with large datasets collected from masses of sensors.

Implementation.
Our approach is implemented and takes the form of a plugin for the Eclipse IDE[2]. The plugin comprises a code generator, which currently produces programming support for the Apache Hadoop platform[3].

Validation.
Our implementation is validated with an experiment that runs application computations over a large dataset of synthetic sensor readings. The experiment demonstrates that programming frameworks generated by our approach exhibit scalable behavior.

II. Background & Case Study

In this section, we provide a brief introduction of the DiaSwarm language [9] dedicated to development of orchestrating applications. DiaSwarm is a declarative domain-specific design language, which follows the Sense/Compute/Control (SCC) paradigm promoted by Taylor *et al.* [12]. DiaSwarm provides high-level, declarative constructs to allow developers to deal with sensors and actuators at design time, prior to programming the application. Application design is processed by a compiler, which generates support for the developer that takes the form of a programming framework [13]. The generated programming framework reflects application design and covers domain-specific functionalities, such as service discovery, data gathering, component interaction and data processing. These dimensions are fully administered by the framework to allow developers to concentrate on the application logic.

Application design takes the form of a directed acyclic graph (DAG) comprising devices (*i.e.,* sensors and actuators) and application components, namely, contexts and controllers. Context components receive data from sensors via device sources. They refine raw data into application values and may publish these values to controller components. Controllers determine the devices that need to be actuated, as well as the type of action that needs to be triggered.

A. Case study

We illustrate the salient features of DiaSwarm with a smart city application, which monitors the occupancy of parking lots to guide cars to available parking spaces. The application collects data from presence sensors, which are buried under the ground and determine availability of parking spaces via magnetic field variations. The application provides drivers with the number of available parking spaces for each parking lot in the city. This information is displayed on screens at the entrance of parking lots. The application also suggests parking lots to drivers entering the city to optimize the flow of traffic. Finally, the application determines the average occupancy level of each parking lot in 24 hours. The occupancy level is provided to parking managers via messages.

Fig. 1 presents a graphical view of the parking management application in SCC. The `PresenceSensor` device produces values via the `presence` source to the subscribed context components, namely `ParkingAvailability`, `ParkingUsagePattern` and `AverageOccupancy`. The `ParkingAvailability` context computes the number of available parking spaces in parking lots and publishes these values at regular intervals to the `ParkingEntrancePanel` controller, which in turn triggers the `update` action to refresh the number of available parking spaces on entrance screens. Parking suggestions for drivers are computed by the `ParkingSuggestion` context, which is invoked every time the `ParkingAvailability` context publishes a value. In this case, the computation carried out by `ParkingSuggestion` context requires also data from the `ParkingUsagePattern` context. The resulting suggestions are published to the `CityEntrancePanelController`, which refreshes these suggestions on entrance panels. The average occupancy level functionality is designed in a similar fashion with the exception of providing computations over a 24-hour period (*i.e.,* `AverageOccupancy` context).

B. Preliminaries

Let us now briefly present the salient features of DiaSwarm declarations through fragments of the design of our case study, displayed in Fig. 2. Note that we omit details on controller components and actuators. The complete design for the parking management application and further information on DiaSwarm can be found on our website.[4]

Service discovery. DiaSwarm service discovery is part of the design phase. The language provides application-specific high-level constructs for discovering objects in the large. The `grouped by` clause allows sensor data to be presented

[2] `http://eclipse.org/`
[3] `http://hadoop.apache.org/`
[4] `http://phoenix.inria.fr/software/diaswarm`

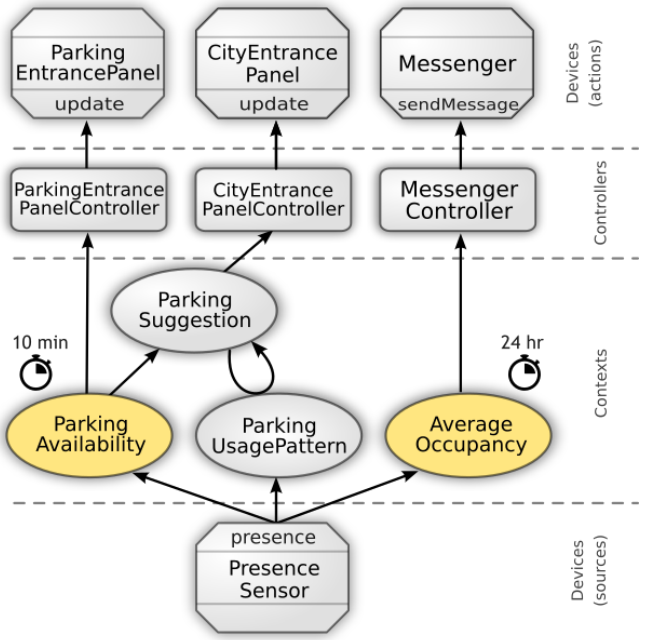

Figure 1: The graphical view of the parking management application.

```
1  context ParkingAvailability as Availability[] {
2    when periodic presence from PresenceSensor <10 min>
3    grouped by parkingLot
4    with map as Boolean reduce as Integer
5    always publish;
6  }

8  context AverageOccupancy as ParkingOccupancy[] {
9    when periodic presence from PresenceSensor <10 min>
10   grouped by parkingLot every <24 hr>
11   with map as Presence reduce as Integer
12   always publish;
13 }

15 device PresenceSensor {
16   attribute parkingLot as ParkingLotEnum;
17   source presence as Boolean;
18 }
19
20 structure Presence {
21   presence as Boolean;
22   time as String;
23 }
```

Figure 2: Excerpt of the parking management application design in DiaSwarm.

to applications through subsets of interest. In the case of the, `ParkingAvailability` context, parking spaces are gathered together in parking lots, as shown in line 3. Similarly, in line 10, the `AverageOccupancy` context groups presence values by parking lots and computes average occupancy over 24 hours.

Data gathering. DiaSwarm provides three data delivery models, inspired by the domain of wireless sensor networks [14], namely periodic, event-driven and query driven. For example, in lines 2 and 9 both `ParkingAvailability` and `AverageOccupancy` contexts require presence measurements to be provided every 10 minutes. Thus, according to this declaration both context components will be activated every 10 minutes with presence values. Furthermore, the event-driven model provides data to context components upon an event of interest (*e.g.,* intrusion). The query-driven model allows a context to request data from devices and other contexts.

Programming frameworks. To enforce domain-specific functionalities (*e.g.,* service discovery) during programming, Java programming frameworks are produced by a compiler from DiaSwarm designs. These frameworks provide an abstract class for each component, which in turn requires developers to implement components by subclassing every abstract class.

C. Data processing

Although high level, the DiaSwarm declarations suggest data processing models. Specifically, an application is reactive and consists of chains of component activations. A chain is executed when its initial activation condition holds (*e.g.,* a sensor publishes data), regardless of the delivery model. The execution of a chain ends if one or more actuators are invoked or a component does not publish any value. Additionally, when a component declaration groups values (*e.g.,* `grouped by parkingLot`), it will process a sequence of values, indexed by the grouping attribute (*i.e.,* `parkingLot`). For example, in the `ParkingAvailability` component, the processing will receive a list of available parking spaces, indexed by parking lot identifiers (*i.e.,* `ParkingLotEnum`). Additionally, this construct allows values to be accumulated over a period of time, as illustrated by the `AverageOccupancy` context (line 8). The declaration in line 10 allows presence values, not only to be grouped by `parkingLot`, but also to be accumulated over a 24-hour period (keyword `every`).

III. Exposing Parallelism

The large amount of data collected from sensors calls for efficient processing strategies. We now examine how an application design influences the way data are processed. This study allows us to propose extensions to DiaSwarm and novel treatments of declarations to generate efficient parallel processing of large-scale datasets.

A. MapReduce

Our aim is to put in synergy design and programming by leveraging design declarations to expose parallelism and allow efficient processing strategies to be implemented. An ideal case study is the `grouped by` directive because it partitions a large set of gathered data and exposes a processing strategy that matches the MapReduce programming model.

```
1  public class ParkingAvailability extends AbstractParkingAvailability
2                          implements MapReduce<ParkingLotEnum, Boolean, ParkingLotEnum, Boolean, ParkingLotEnum, Integer> {
3       @Override
4       public void map(ParkingLotEnum parkingLot, Boolean presence, MapCollector<ParkingLotEnum, Boolean> collector) {
5            if(!presence)
6                collector.emitMap(parkingLot, true);
7       }
8
9       @Override
10      public void reduce(ParkingLotEnum parkingLot,
11                         List<Boolean> values, ReduceCollector<ParkingLotEnum, Integer> collector) {
12           int sum = 0;
13           for (int i = 0; i < values.size(); i++) {
14               sum++;
15           }
16
17           collector.emitReduce(parkingLot, sum);
18      }
19
20      @Override
21      protected List<Availability> onPeriodicPresence(Map<ParkingLotEnum, Integer> presenceByParkingLot) {
22           List<Availability> availabilityList = new ArrayList<Availability>();
23
24           for(Entry<ParkingLotEnum, Integer> parkingLot : presenceByParkingLot.entrySet()) {
25               Availability availability = new Availability(parkingLot.getKey(), parkingLot.getValue());
26               availabilityList.add(availability);
27           }
28
29           return availabilityList;
30      }
31 }
```

Figure 3: An implementation of the ParkingAvailability context with MapReduce.

Indeed, this programming model is dedicated to processing large datasets in a massively parallel manner [10], [11]. It requires processing to be split into two phases: Map and Reduce. Following our approach, data processing needs to be reflected in the design phase. This is done by extending the `grouped by` directive with an optional clause that specifies what types of values are produced by both the Map and Reduce phases. This is illustrated in Fig. 2, where the `ParkingAvailability` declaration includes a MapReduce clause that declares the Map phase to produce `Boolean` values and the Reduce phase to produce `Integer` values.

The DiaSwarm compiler generates a programming framework that requires the developer to provide an implementation for both the Map and Reduce phases of the data processing. As shown in Fig. 3, this is done by implementing `map` and `reduce` methods declared in the generated MapReduce interface. In conformance with the MapReduce model, the Map function is passed a key and a value, which correspond to the parking lot identifier (*i.e.,* the attribute of the `grouped by` directive) and an availability status, provided by the corresponding sensor. The `emitMap` method is invoked to produce each key/value pair result of the Map phase. The framework-generated code groups the results of the Map phase into a list that is then passed to the Reduce phase. This phase sums up the set of values associated with a given intermediate key and, subsequently, emits the availability of a parking lot (`emitReduce`). The data resulting from the MapReduce computation are presented to the developer in the form of a map (line 21). The `onPeriodicPresence` method (line 21 to 30) wraps data resulting from the MapReduce process into the `availabilityList` sequence (line 26), which is returned to subscribed components (*i.e.,* `ParkingEntrancePanelController`, `ParkingSuggestion`).

Although our example involves simple processing, in practice, our design-driven generative approach reduces programming efforts by automatically generating application-specific MapReduce programming frameworks. Furthermore, the generated code keeps the development process straightforward since it prevents specificities of the MapReduce implementation (job scheduling/configuration/execution, distributed file system, APIs, *etc.*) to percolate into the application logic.

B. Integrating Hadoop

Our design-driven development approach facilitates the processing of large datasets collected from sensor infrastructures by providing the developer with a customized framework, following the MapReduce programming model. In this section, we show how generative programming is used to produce support for combining an orchestrating application with an actual implementation of MapReduce, namely Hadoop.

Apache Hadoop is an open source implementation of the MapReduce paradigm, which has gained increasing attention

```
1  public class ParkingAvailabilityJob extends Configured implements Tool {
2
3      public static class ParkingAvailabilityMap extends MapReduceBase
4                                              implements Mapper<LongWritable, Text, Text, BooleanWritable> {
5          @Override
6          public void map(LongWritable key, Text value, OutputCollector<Text, BooleanWritable> output, Reporter reporter) {
7              jobLauncher.doMap(key, value, output);
8          }
9      }
10
11     public static class ParkingAvailabilityReduce extends MapReduceBase
12                                             implements Reducer<Text, BooleanWritable, Text, IntWritable> {
13         @Override
14         public void reduce(Text key, Iterator<BooleanWritable> values, OutputCollector<Text, IntWritable> output, Reporter
               reporter) {
15             jobLauncher.doReduce(key, values, output);
16         }
17     }
18
19     @Override
20     public int run(String[] args) {
21         JobConf conf = new JobConf(getConf(), ParkingAvailabilityJob.class);
22         conf.setInputFormat(TextInputFormat.class);
23         // Remaining configuration
24     }
25 }
```

Figure 4: An example of the generated Hadoop MapReduce program for the ParkingAvailability context.

over the last years and is currently being used by a number of companies, including IBM, LinkedIn, Facebook and Google [15]. In our approach, our compiler generates a MapReduce program that relies on the Hadoop framework. Furthermore, this MapReduce program defines default configuration parameters that enable a job to be executed in Hadoop.

Let us describe how this is achieved, by examining the code automatically generated for the `ParkingAvailability` context, shown in Fig. 5. The `ParkingAvailabilityJob` class defines a Hadoop MapReduce program, which comprises the definition of both the `map` and `reduce` methods along with code related to the job configuration and execution. Both the Map function and the Reduce function are implemented by overriding the `map` and `reduce` methods of the respective `Mapper` and `Reducer` interfaces. Typically, when using the Hadoop MapReduce library, the definition of the `map` and `reduce` methods resides in the MapReduce program. In this case, however, the implementation of these operations has already been provided by the developer in the `ParkingAvailability` class. The MapReduce program invokes the user-defined `map` and `reduce` methods via the `ParkingAvailabilityParser` class, which keeps an instance of the `ParkingAvailability` context. `ParkingAvailabilityParser` interprets input data of the MapReduce program as corresponding DiaSwarm types and invokes the required `map`/`reduce` method. Consequently, results from the user-defined `map`/`reduce` method are translated to the MapReduce program and submitted via its output collector.

Fig. 4 shows the `ParkingAvailabilityJob` class, which

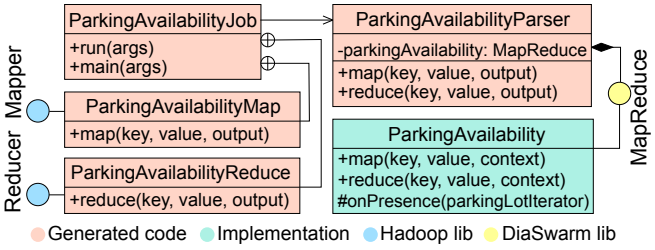

Figure 5: The generated support for integrating Apache Hadoop.

defines the MapReduce program for the `ParkingAvailability` context. The compiler generates a minimal MapReduce program for every context declared as MapReduce at design time. The type of input data for a generated MapReduce program is defined by the input format, which defaults to `TextInputFormat` (line 22). In our approach, sensor data is stored in the JSON format. In our case study, each presence status delivered to the application is converted to JSON and occupies precisely one line in the resulting dataset. Furthermore, each presence entry is defined by the timestamp of the event, device attributes (*i.e.,* id, parking lot) and the presence source. `TextInputFormat` fits such usage since it splits the input dataset to provide the Map function with one line of text (*i.e.,* one JSON entry) at a time. In a MapReduce program, any key or value type implements the `Writable` interface, which allows Hadoop to serialize objects for transmission over the network [16]. To facilitate the development of MapReduce programs, Hadoop already provides `Writable` wrapper classes for the majority

of Java primitives (*e.g.,* `boolean` → `BooleanWritable`). In addition, developers may provide custom datatypes by defining classes implementing the `Writable` interface. At this stage, design declarations are of great importance since they allow the compiler to interpret key and value types of the resulting MapReduce program. For instance, as shown in Fig. 2, the `ParkingAvailability` context declares the output value type of the Map function as `Boolean` (line 4). As a result, the compiler matches the `Boolean` data type with the corresponding `BooleanWritable` wrapper class (Fig. 4, line 6). Moreover, an enumeration is interpreted as a string and matched with the `Text` wrapper class (Fig. 4, line 6). Finally, design declarations using complex data types result in the generation of a custom wrapper class, which implements the `Writable` interface and reflects the entire structure of the datatype.

The execution of a MapReduce program depends upon the data delivery model underlying the interaction between sensors (devices) and the application logic (contexts). In our case study, the `ParkingAvailability` context declares that data must be gathered from presence sensors in a 10-minute time window according to a periodic delivery model (Fig. 2, line 2). Data processing takes place when the time window elapses; that is, every 10 minutes, for our case study. At runtime, this job is executed with respect to the gathered sensed data and produces a result. The orchestrating application recovers the result, which is passed to the context via its callback method (*e.g.,* `onPeriodicPresence` for `ParkingAvailability`).

C. Other data processing methods

Nowadays, the field of Big Data is attracting much attention from research and industry. The tool-development efforts devoted to dealing with rapidly emerging sources of big data result in an abundance of open-source projects [17]. Apache Hadoop is a widely-used tool to deal with large-scale datasets because it provides a reliable and scalable solution, maintained by a large community of developers. Hadoop is a batch-processing tool, typically used to analyze log files of large-scale systems, collected over a long period of time. The order of magnitude of these systems may range from hundreds of gigabytes to petabytes and, possibly terabytes. Apache Spark [18] is an alternative large-scale, data processing tool, which is gaining popularity due to its promise to outperform Hadoop by 10x [19]. Spark is an in-memory, data processing framework, which builds upon fault tolerant abstractions, manipulated using a rich set of operators, called Resilient Distributed Datasets (RDDs) [20]. In contrast with batch-processing tools, Apache Storm [21] primarily targets the processing of unbounded streams of data. Storm is an example of a CEP [22] system, where data flow through a network of transformation entities. An application topology forms a directed acyclic graph, where stream sources (spouts) flow data to sinks (bolts); it implements a single transformation on the provided stream. In the context of large-scale orchestration, the power of batch-processing tools can be leveraged to analyze long-term datasets for trends in the usage of the city's infrastructure (*e.g.,* parking lots) and to identify structural degradation (*e.g.,* buildings, bridges). Stream processing tools, on the other hand, are best-suited to deal with high-frequency sensor readings, which typically involve tracking applications (*e.g.,* vehicle position, parking place availability). In the future, we intend to extend the parallel data-processing compiler to integrate both Spark and Storm, allowing developers to choose the right tool for their project.

IV. Experimental Evaluation

To assess our approach, we have conducted a series of tests to examine the overall behavior of the MapReduce programming model for processing large amounts of sensor data. To do so, we developed a prototype of the parking management system, with Hadoop as the target platform, and analyzed the scalability of our approach using various datasets. In addition, we evaluated the design of the application and observed how specific design choices may impact the overall performance of an orchestrating application.

A. Experimental setup

The experimentation focuses on the average parking occupancy feature of our case study. The `AverageOccupancy` context processes sensor data acquired over a 24-hour period, calculates the average occupancy of a parking lot, and notifies the parking manager via a `Messenger` device.

Machines. The experiment was carried out on a cluster of 12 nodes running within a private Eucalyptus [23] cloud. Each node in the cloud corresponds to a m2.xlarge type virtual machine instance with 2 CPUs, 2GB of RAM and 10GB of disk space. Every instance ran the DataStax Enterprise 4.6.1 [24] image, which is a big data platform leveraging tools such as Apache Hadoop and Apache Spark.

Datasets. We generated synthetic datasets to simulate a city's sensor infrastructure for the parking management system. Each dataset contains sensor data, indicating parking space occupancy, which is emitted every 10 minutes over 24 hours (*i.e.,* 144 measurements per sensor). We generated datasets for different sensor infrastructures, ranging from 10 000 to 200 000 sensors per dataset, thus testing the MapReduce program with datasets including up to 28 800 000 input records.

B. Experimental results

Scalability. Fig. 6 shows the performance of our parking management program. We compare its execution time with respect to 3 cluster setups – one, six and twelve nodes – and an increasing input dataset size. As can be expected, the execution time of the one-node setup increases the

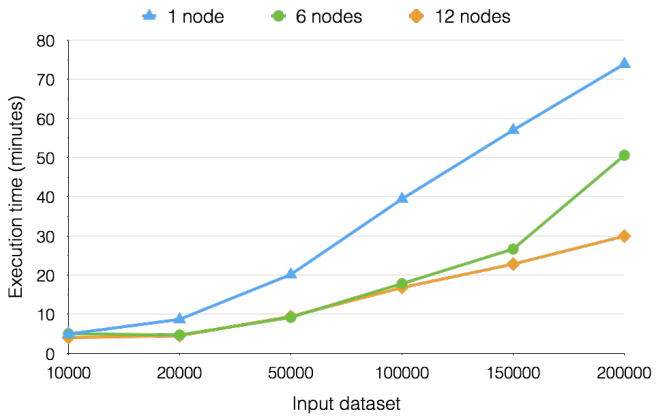

Figure 6: Performance comparison between different cluster setups.

fastest, compared to the six and twelve node setups. The six and twelve node setups perform at par for the smallest dataset sizes (from 10 000 to 50 000 sensors) because their computing power is under-used. As the size of the datasets increases, the performance of these two setups gradually separate, showing better performance for the twelve-node setup. These preliminary results show that our compiler generates MapReduce implementations that attain expected scalability. Furthermore, these results demonstrate that declarations at the design level can benefit performance by driving compilation strategies, such as parallelization in our case study. This is achieved by introducing high-level insights (MapReduce constructs) in DiaSwarm.

Optimization through design. Beyond significantly improving the execution time of an orchestrating application, Hadoop opens up further optimization opportunities at the design level. For instance, in our case study, the `AverageOccupancy` context processes a dataset of presence values to produce the average occupancy of each parking lot for the last 24 hours. A closer look at the application design reveals that the computation provided by the `AverageOccupancy` context could be achieved by leveraging the computation of the `ParkingAvailability` context. The computed availability of parking spaces could thus be provided to the `AverageOccupancy` context at regular intervals, defined by the data delivery contract (*i.e.,* <10 min>) of the `ParkingAvailability` context. As a result, the `AverageOccupancy` context would use the provided data to calculate a cumulated moving average over the period of 24 hours.

The suggested design adjustments are depicted in Fig. 7. As can be noticed, the design of the application remains straightforward. More importantly, this design prevents sensor readings from being processed multiple times: the `AverageOccupancy` context factorizes the computations performed by the `ParkingAvailability` context. This caching strategy reduces the total time and resources the application requires for data processing. In fact, as shown in Fig. 7, the

```
1 context AverageOccupancy as ParkingOccupancy[] {
2   when provided ParkingAvailability // replaces line 9,
        Fig. 2
3   grouped every <24 hr> // replaces line 10, Fig. 2
4   always publish;
5 }
```

Figure 7: The ParkingAvailability context factorizing the computation performed by AverageOccupancy.

computation performed by the `AverageOccupancy` context no longer involves processing of a large dataset on a cluster (hence the MapReduce clause is omitted).

This major optimization also has a direct impact on application upkeep costs, since nowadays companies delegate processing of large datasets to cloud computing platforms (*e.g.,* Amazon Web Services) with a time-of-use pricing model.

V. Related Work

In this section, we examine existing approaches that address the development of applications orchestrating sensors. We consider approaches from domains where orchestration of sensors is a common concern. Furthermore, we highlight the differences between our approach and large-scale data processing support.

Internet of Things (IoT). Patel *et al.* propose a multi-stage, model-driven approach, dedicated to the development of IoT applications [25]. This approach provides support at different stages of the development process. At design time, the approach offers a set of customizable modeling languages for the specification of an application. The approach is complemented by code generation and task-mapping techniques for the deployment of node-level code onto devices. Even though this approach is aimed to facilitate the development process through guidance, Patel *et al.* do not provide details regarding the size of sensed data that are gathered and processed. They do not discuss what support is generated to facilitate the programming process. This approach does not address how masses of sensors are handled, nor does it present performance measurements to assess how it scales up for large datasets.

Pervasive computing. The domain of pervasive computing offers a number of approaches targeting the development of orchestrating applications. PervML [26] is a model-driven development approach that provides a conceptual framework for context-aware applications. The various aspects of a pervasive computing application are modeled by different types of UML diagrams. Dey *et al.* propose the Context Toolkit [27] that provides the programmer with building blocks to mediate between the contextual aspects of the environment and the application. Olympus goes beyond middleware in providing a programming framework dedicated to the development of pervasive computing systems [28].

Because it is based on a domain-specific framework, Olympus raises the level of abstraction and facilitates the development of applications. DiaSuite takes these approaches further by introducing a design language dedicated to the Sense/Compute/Control paradigm [29], [30]. A design is used to generate a dedicated programming framework that guides, restricts, and supports the implementation phase.

All the above-mentioned approaches have been designed for the orchestration of objects in the small (*i.e.,* offices, buildings, *etc.*). They do not address challenges arising with large-scale infrastructures and do not provide strategies to tackle data-intensive processing.

Wireless sensor networks (WSN). Gupta *et al.* propose sMapReduce [31], a programming pattern inspired by the MapReduce programming model for mapping application behavior onto a sensor network and enabling complex data aggregation. sMapReduce divides the network-level user program into sMap and Reduce functions; this strategy respectively associates a behavior to sensor nodes and executes data aggregation over the network. Compared to our approach, sMapReduce remains lower-level since it provides network-level programming abstractions and introduces the network topology in computations.

Often, programming applications for WSNs is done at a low level, requiring the developer to have extensive knowledge about the underlying layers (network, hardware, OS). Mottola and Picco [32] surveyed a number of programming approaches for WSNs aimed to facilitate the programming of layers underlying applications; these approaches target sensor nodes, communication operations, routing strategies, *etc.* These works are complementary to ours in that they provide high-level abstractions that can be used by our compiler to target frameworks for WSNs. However, they do not provide support dedicated to dealing with large datasets produced from massive-scale sensor infrastructures.

Large-scale data processing. Apache Pig [6] and Apache Hive [7] are widely used as high-level platforms for analyzing large-scale datasets. These platforms provide SQL-like declarative query languages (*i.e.,* PigLatin & HiveQL) to express data analysis programs. These tools are well-suited for offline data analysis, but require some effort for running scripts from application code (*e.g.,* setting up a connection with a JDBC server). Sawzall [33] used by Google is a high-level scripting language for automating analyses on large data sets on top of the MapReduce execution model. Sawzall is not publicly available but is reported to improve the programming significantly, compared to C++ programming of MapReduce. High-level language libraries, such as FlumeJava [8], provide high-level abstractions dedicated to parallel processing; they provide support for user-defined functions, compared to SQL-like approaches.

Compared to the above-mentioned supports, our approach integrates, at the design level, two domain-specific fundamental dimensions: large-scale orchestration of sensors and large-scale data processing. The integrated nature of our approach allows developers to easily combine results from various computations. The design-driven nature of our approach is supported by high-level declarations, exposing such domain-specific information as service discovery and data delivery. Declarations are analyzed to determine data and control flow information, which in turn, is used to generate efficient, parallel-data processing frameworks.

VI. Conclusion and Future Work

We have proposed a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of sensed data. Our new approach provides the developer with design declarations expressing when and where data processing occurs. A compiler takes an application design as input and produces a programming framework based on the MapReduce programming model. The generated framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

We have demonstrated that our approach creates synergy between design and programming, allowing seamless introduction of high-performance computing strategies, as illustrated by the MapReduce programming model. We illustrated our approach with a case study of a parking management system. This case study was used to conduct an experiment on Apache Hadoop, demonstrating how our design-driven approach can be leveraged to parallelize the processing of large datasets and obtain significant speedups.

In the future, we intend to support the processing of unbounded streams of data, typical of sensors. Our declarative approach will allow us to design orchestrating applications that mix the processing of both large datasets and unbounded data streams, allowing us to abstract away these aspects.

References

[1] Y. Mizuno and N. Odake, "Current Status of Smart Systems and Case Studies of Privacy Protection Platform for Smart City in Japan," in *2015 Portland International Conference on Management of Engineering and Technology (PICMET)*, Aug 2015, pp. 612–624.

[2] M. Naphade, G. Banavar, C. Harrison, J. Paraszczak, and R. Morris, "Smarter Cities and Their Innovation Challenges," *Computer*, vol. 44, no. 6, pp. 32–39, June 2011.

[3] Libelium, "Smart City project in Santander to monitor Parking Free Slots," March 2016. [Online]. Available: http://www.libelium.com/smart_santander_parking_smart_city.

[4] Worldsensing SL, "Worldsensing and SIGFOX announce the world's largest Intelligent Parking deployment with Micronet, the SIGFOX Network Operator for Russia," March 2016. [Online]. Available: http://www.worldsensing.com/news-press/press-release-worldsensing-and-sigfox-announce-the-worlds-

largest-intelligent-parking-deployment-with-micronet-the-sigfox-network-operator-for-russia.html.

[5] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, December 2012.

[6] The Apache Software Foundation, "Apache Pig," March 2016. [Online]. Available: https://pig.apache.org.

[7] The Apache Software Foundation , "Apache Hive," March 2016. [Online]. Available: https://hive.apache.org.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, Efficient Data-Parallel Pipelines," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 2010, pp. 363–375.

[9] M. Kabáč and C. Consel, "Orchestrating Masses of Sensors: A Design-Driven Development Approach," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015)*. ACM, 2015, pp. 117–120.

[10] R. Lämmel, "Google's MapReduce programming model - Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, Oct. 2008.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[12] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[13] M. Fayad and D. C. Schmidt, "Object-Oriented Application Frameworks," *Commun. ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997.

[14] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A Taxonomy of Wireless Micro-Sensor Network Models," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 2, pp. 28–36, Apr. 2002.

[15] The Apache Software Foundation, "Hadoop Wiki PoweredBy," March 2016. [Online]. Available: http://wiki.apache.org/hadoop/PoweredBy.

[16] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[17] The Apache Software Foundation, "Projects Directory," March 2016. [Online]. Available: https://projects.apache.org/projects.html?category#big-data.

[18] The Apache Software Foundation , "Apache Spark," March 2016. [Online]. Available: http://spark.apache.org.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 2010, pp. 10–10.

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 2012, pp. 2–2.

[21] The Apache Software Foundation, "Apache Storm," March 2016. [Online]. Available: http://storm.apache.org.

[22] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, 2012.

[23] Hewlett-Packard, "HP Helion Eucalyptus," March 2016. [Online]. Available: http://www.eucalyptus.com.

[24] DataStax, "DataStax Enterprise," March 2016. [Online]. Available: http://www.datastax.com.

[25] P. Patel, A. Pathak, D. Cassou, and V. Issarny, "Enabling High-Level Application Development in the Internet of Things," in *S-CUBE'13: 4th International Conference on Sensor Systems and Software*, Jun. 2013.

[26] E. Serral, P. Valderas, and V. Pelechano, "Towards the Model Driven Development of Context-aware Pervasive Systems," *Pervasive Mob. Comput.*, vol. 6, no. 2, pp. 254–280, Apr. 2010.

[27] A. K. Dey, G. D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications," *Hum.-Comput. Interact.*, vol. 16, no. 2, pp. 97–166, Dec. 2001.

[28] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, "Olympus: A High-Level Programming Model for Pervasive Computing Environments," in *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications (PERCOM '05)*. IEEE Computer Society, March 2005, pp. 7–16.

[29] D. Cassou, E. Balland, C. Consel, and J. Lawall, "Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, 2011, pp. 431–440.

[30] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel, "DiaSuite: a Tool Suite To Develop Sense/Compute/Control Applications," *Science of Computer Programming*, vol. 79, pp. 39–51, Jan. 2014.

[31] V. Gupta, E. Tovar, L. M. Pinho, J. Kim, K. Lakshmanan, and R. Rajkumar, "sMapReduce: A Programming Pattern for Wireless Sensor Networks," in *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications (SESENA '11)*. ACM, 2011, pp. 37–42.

[32] L. Mottola and G. P. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, 2011.

[33] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, Oct. 2005.